מיונים א':

מיון ערימה (HeapSort)
QuickSort

חומר קריאה לשיעור זה

Chapter 7 - Heapsort
Chapter 8 - Quicksort

מיון (Sorting)

<u>קלט</u>: מערך בן n מספרים.
<u>פלט</u>: מערך ובו המספרים מאוחסנים בסדר עולה (או יורד).

שיטות מיון נאיביות דורשות זמן $\Theta(n^2)$.
<u>לדוגמא</u>, החלפת איברים סמוכים (Bubble Sort):

```
void BubbleSort(int* A, int n){
        for (i = 0; i < n-1;  i++)
                for (j = n-2;  j >= i;  j--)
                        if ( a[j]  >  a[j+1])
                                swap(&a[j], &a[j+1]);
}
```

1	1	1	1
2	2	2	2
3	3	3	3
7	5	5	5
5	7	6	6
9	6	7	7
6	9	8	8
8	8	9	9

בכל שלב האיבר הקל ביותר שלא במקומו מבעבע למעלה

<u>הערה</u>: בד"כ ממיינים רשומות לפי המפתח שלהם ולא סתם מספרים. לשם בחינת שיטות המיון מספיק לבחון את מיון המפתחות עצמם. כאשר הרשומות מכילות אינפורמציה רבה, אלגוריתם מיון משנה בכל שלב מצביעים לרשומות ולא את הרשומות עצמן.

תור עדיפויות/ ערימה

<u>תור עדיפויות</u> (או ערימה - Heap) הוא מבנה נתונים המוגדר ע"י הפעולות הבאות:

MakeHeap(Q) צור ערימה ריקה.

Insert(x, Q) הכנס רשומה x לערימה.

Max(Q) הדפס את הרשומה עם המפתח הגדול ביותר בערימה.

del_max(Q) הוצא את הרשומה עם המפתח הגדול ביותר בערימה.

המבנה נקרא תור עדיפויות לאור אחד השימושים הנפוצים שלו: כל רשומה מגדירה משימה (job) עם דרגת עדיפות. המשימה הבאה לבצוע היא המשימה בעלת העדיפות הגבוהה ביותר. מבנה זה מכליל תור רגיל שבו עדיפות גבוהה ניתנת לפי סדר ההכנסה.

<u>מימוש נאיבי לערימה</u>

עץ חיפוש מאוזן. כל פעולת הכנסה והוצאה בזמן $O(\log n)$.

HeapSort - מיון בעזרת תור עדיפויות\ערימה

<u>HeapSort - מיון בעזרת ערימה</u>

1. <u>אתחול</u>: make_heap(Q)
2. לכל x בקלט: insert(x, Q)
3. כל עוד התור אינו ריק:

Output max(Q);

del_max(Q)

<u>במימוש הנאיבי באמצעות עץ חיפוש:</u>

כל פעולת הכנסה, הוצאה, חיפוש בזמן $O(\log n)$.

סה"כ זמן $O(n \log n)$.

בפעולות שהגדרנו, בניית ערימה מ- n איברים יכולה לקחת $\Theta(n \log n)$ זמן כתלות בסדר ההכנסה.

נגדיר פעולת make_heap($Q, x_1, \ldots, x_n$) היוצרת ערימה בת n איברים (פעולה זו מחליפה את שני הצעדים הראשונים באלגוריתם המיון הרשום מעלה).

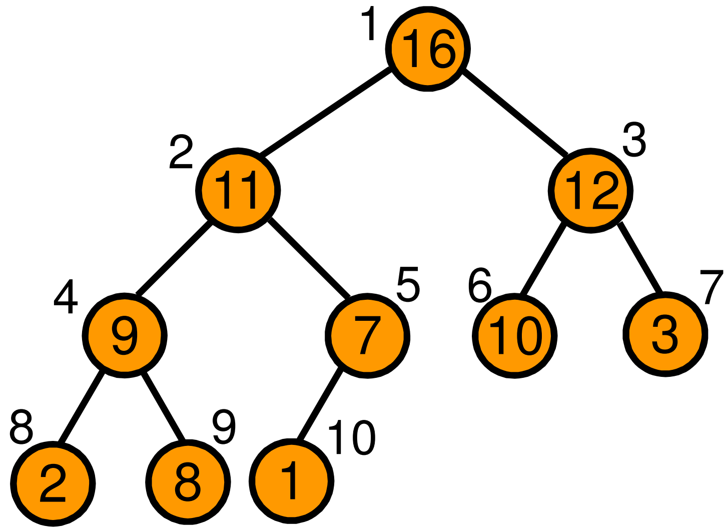
נראה שמימוש יעיל של פעולה זו דורש $O(n)$ זמן.

מימוש ערימה

<u>מימוש ערימה בעזרת עץ בינרי כמעט שלם</u>

המפתח של הורה גדול או שווה ממפתחות ילדיו (תנאי זה נקרא **תכונת הערימה**).
הסדר בין הילדים אינו מוגבל.

<u>דוגמא:</u>



כזכור מהרצאות קודמות, עץ כמעט שלם ניתן לייצג גם במערך:

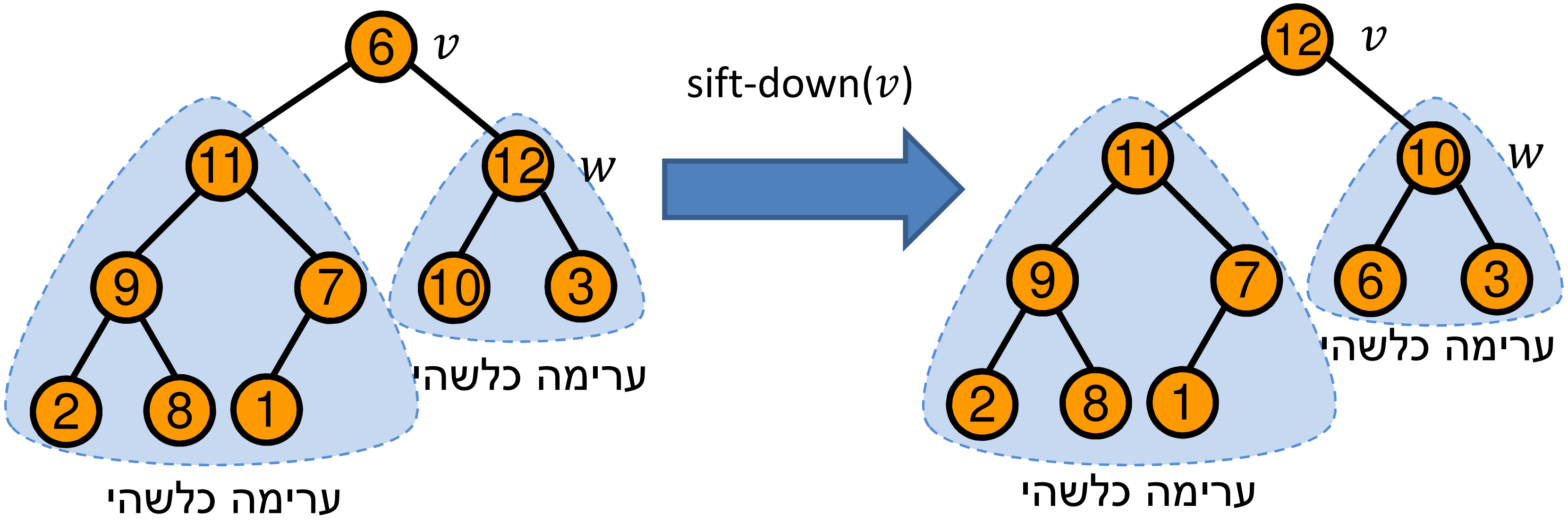
1	2	3	4	5	6	7	8	9	10
16	11	12	9	7	10	3	2	8	1

כאשר ההורה של צומת i הוא הצומת $\lfloor i/2 \rfloor$, בן שמאלי של צומת i הוא הצומת $2i$
ובן ימני הוא הצומת $2i+1$.

פרוצדורת עזר (sift-down)

בזמן הוצאה והכנסה יש לשמור על תכונת הערימה. לשם כך נשתמש בפרוצדורה ההופכת עץ בינרי כמעט שלם אשר עבורו מופרת תכונת הערימה רק בשורש בחזרה לערימה.

העץ מורכב משורש v המצביע לשתי ערימות:



<u>כיצד נתקן את הערימה?</u>

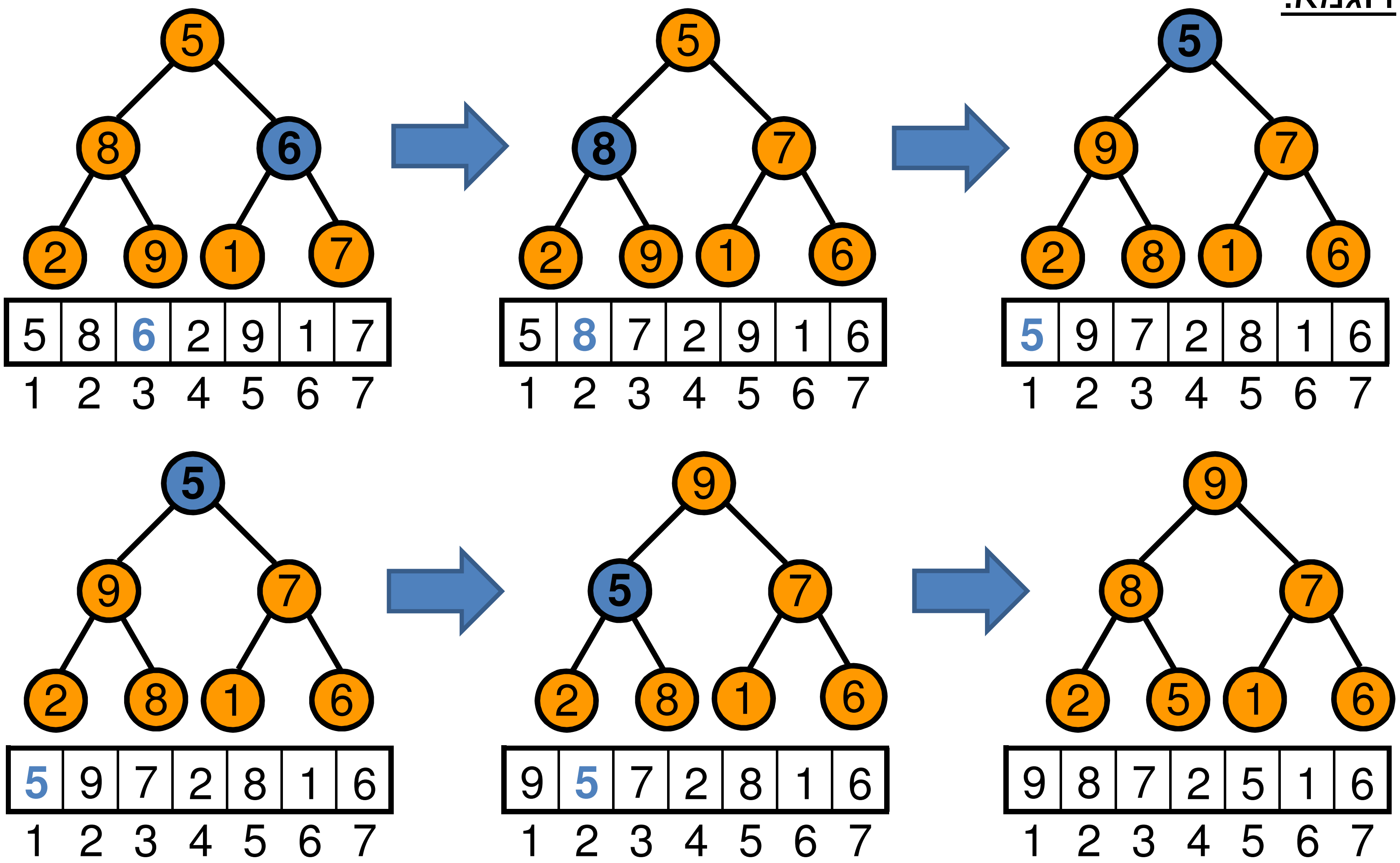
1. אם v עלה, או v גדול משני בניו – סיים (העץ הנתון מהוה ערימה).
2. אחרת, החלף את השורש v עם הבן בעל המפתח המקסימלי w והמשך עם הערימה ששורשה w.

אלגוריתם זה נקרא sift-down(v), "סינון כלפי מטה".

פרוצדורת make-heap

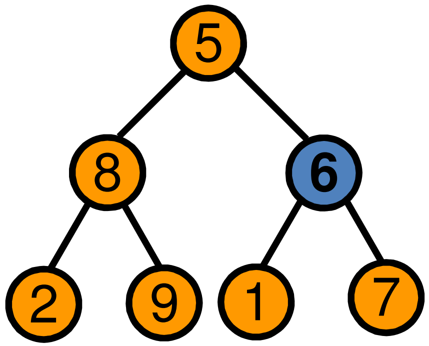
מימוש $make\_heap(T, x_1, \ldots, x_n)$: עבור על כל צמתי העץ הפנימיים כך שנעבור על הבנים לפני הוריהם (למשל בסדר Postorder). לכל צומת בעץ בצע $sift\_down(v)$.

<u>דוגמא:</u>



נכונות פרוצדורת make-heap

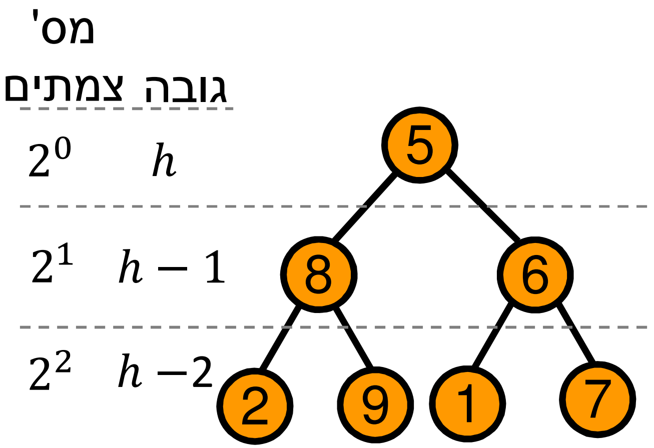
<u>נכונות</u>: לאור סדר בחירת הצמתים בזמן שהאלגוריתם מבצע sift-down(v) הצומת v כבר מצביע על עצים המקיימים את תכונת הערימה. לפיכך לאחר ביצוע sift-down(v), נוצרת ערימה ששורשה v.
נימוק זה נכון לכל צומת, כולל השורש, ולפיכך בסיום מוחזרת ערימה.



<u>זמן הריצה</u> של make-heap נתון ע"י O(סכום הגבהים של כל הצמתים), שכן לכל צומת מבצעים $O(h(v))$ תיקונים כאשר $h(v)$ הוא גובה הצומת v.
ניתוח פשוט של סכום הגבהים נותן חסם גס של $O(n \log n)$, שכן הגובה של כל צומת חסום ע"י $O(\log n)$.
ניתוח מדויק יותר ייקח בחשבון שלרוב הצמתים גובה קטן. ניתוח כזה נותן חסם הדוק של $O(n)$.

ניתוח זמנים של פרוצדורת make-heap

זמן הריצה של make-heap נתון ע"י
סכום הגבהים של כל הצמתים: $O\left(\Sigma_v h(v)\right)$.



<u>משפט</u>: עבור עץ בינרי שלם בגובה h הכולל $n = 2^{h+1} - 1$ צמתים, סכום הגבהים קטן מ- n.

<u>מסקנה</u>: זמן הריצה של Make_Heap על n איברים הוא $O(n)$.

<u>הוכחת המשפט</u>: ישנו צומת בודד בגובה h, 2 צמתים בגובה $h-1$, 2^2 צמתים בגובה $h-2$, ובאופן כללי 2^i צמתים בגובה $h-i$.
לפיכך סכום הגבהים (שנסמנו ב- S) מקיים:

$$S = \sum_{i=0}^{h-1} 2^i(h-i) = \quad 1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \cdots + 2^{h-1} \cdot 1$$

$$2S = \quad 2 \cdot h \quad + 4 \cdot (h-1) + 8 \cdot (h-2) + \cdots + 2^h \cdot 1$$

ע"י חיסור המשוואה הראשונה מהשניה:

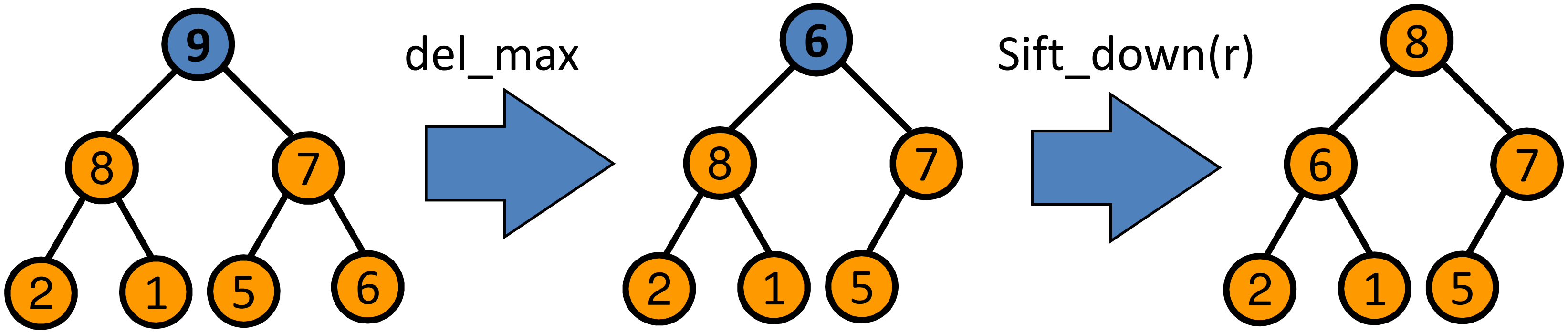
$$S = -h + 2 + 4 + 8 + \cdots + 2^h = -h - 1 + \left(2^{h+1} - 1\right) < n$$

HeapSort

כיצד ממומש $\max(S)$?
הדפס את השורש. זמן: $O(1)$.

כיצד ממומש $\text{del\_max}(S)$?
שים עלה אחרון במקום השורש r. בצע $\text{sift\_down}(r)$. זמן: $O(\log n)$.

דוגמא:



HeapSort המשופר, עם שימוש ב Make_Heap:

```
HeapSort(x_1, ..., x_n){
    make_heap(S, x_1, ..., x_n);
    While ( S ≠ ∅) {
        Output max(S);
        del_max(S);
    }
}
```

דוגמא למיון HeapSort

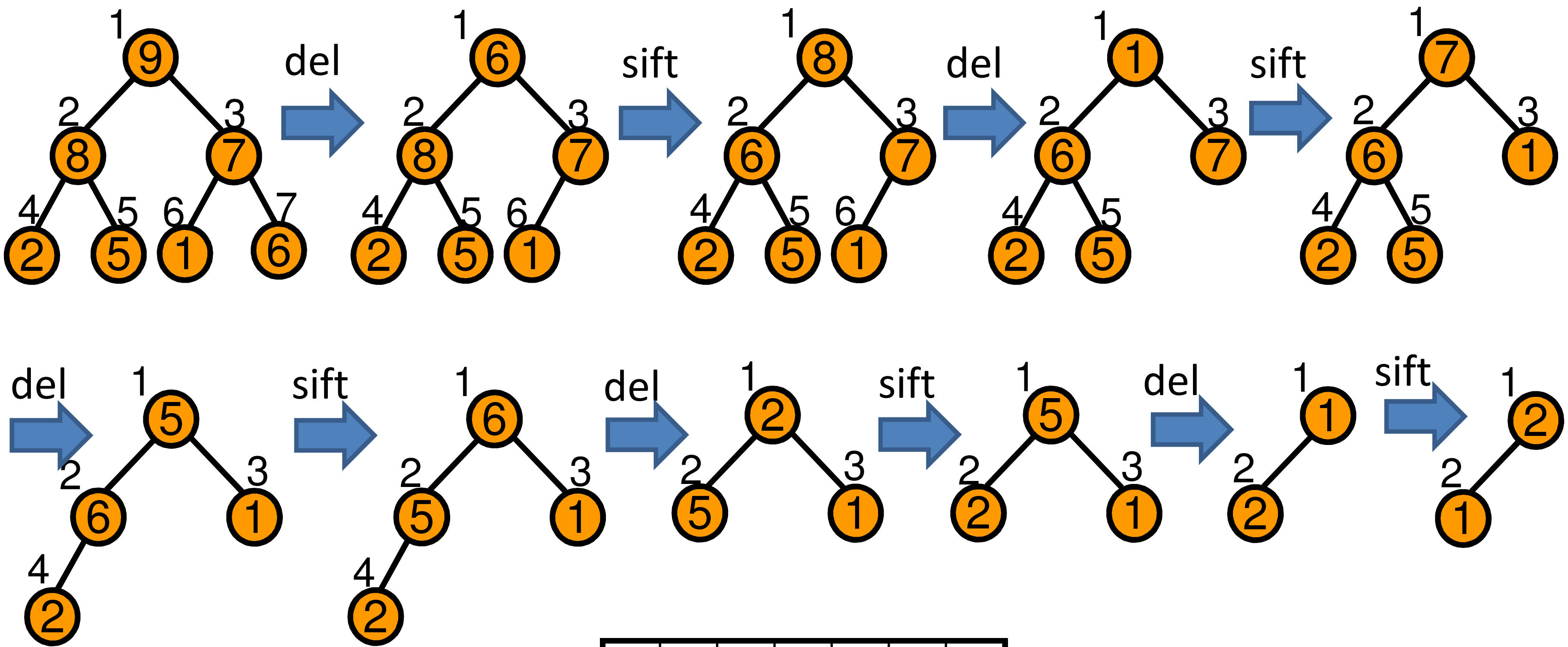
המערך הנתון (כמו בשקף 7):

5	8	6	2	9	1	7

המערך לאחר make_heap (כמו בשקף 7):

9	8	7	2	5	1	6
1	2	3	4	5	6	7

<u>פעולות המיון:</u>



<u>המערך הממוין:</u>

1	2	5	6	7	8	9
1	2	3	4	5	6	7

HeapSort - אנליזת מקום וזמן

<u>סיבוכיות זמן:</u>
סך הזמן הנדרש מורכב מזמן בניית הערימה $O(n)$ בתוספת n פעולות del_max .
כלומר הזמן הכללי הוא:

$$O(n) + O(n\log n) = O(n\log n)$$

<u>סיבוכיות מקום:</u>
פרוצדורת מיון זו פועלת ללא הזדקקות למקום נוסף מעבר למערך המקורי (פרט לתוספת $O(1)$ למשתנים זמניים). המקום הנדרש הוא $O(n)$.

מימוש sift_down לערימת מקסימום

```
sift_down(first, last) {
for (int r = first; r <= last/2; ){
        if (2*r = = last){ /* r has one child at 2*r  */
                if (a[r] < a[2*r])
                        swap(r,2*r);
        }
        else {   /* r has two children at 2*r and 2*r+1  */
                if (a[r]  <  a[2*r] && a[2*r] >=  a[2*r+1]) {
                        swap(a+r, a+2*r);
                        r *= 2;
                }
                else if (a[r] < a[2*r + 1] && a[2*r + 1] >= a[2*r]) {
                        swap(a+r, a+2*r + 1);
                        r = 2*r + 1;
                }
                else break;
}    }    }
```

תוכנית המיון (המשך)

```
heap_sort(n){                                    /* בנית הערימה */
        for (int i = n/2; i > 0; i--)
                sift_down(i,n));
        for (int i = n; i > 0;  i--){            /* המיון עצמו */
                swap(1,i);
                sift_down(1,i-1);
        }
}
```

הערה: בתוכנית זו הפלט ממוין מהקטן לגדול כמודגם בשקפים 7 ו-11.

יתרונות אלגוריתם HeapSort למיון:

- זמן במקרה הגרוע $O(n \log n)$.
- לא דורש מקום עזר.

ערימת מינימום

תור עדיפויות מוגדר לעתים להיות ערימת מינימום, המוגדרת ע"י הפעולות:

$\text{MakeHeap}(Q, x_1, \dots, x_n)$	צור ערימה המכילה את הרשומות $x_1, \dots, x_n$.
$\text{Insert}(x, Q)$	הכנס רשומה x לערימה.
$\text{Min}(Q)$	הדפס את הרשומה עם המפתח הקטן ביותר בערימה.
$\text{del\_min}(Q)$	הוצא את הרשומה עם המפתח הקטן ביותר בערימה.

מימוש ערימת מינימום:
באופן סימטרי למימוש ערימת מקסימום: בעזרת עץ בינרי כמעט שלם כאשר המפתח של הורה קטן או שווה ממפתחות ילדיו.

אלגוריתם HeapSort עם ערימת מינימום:

```
HeapSort(x_1, ..., x_n){
    make_heap(S, x_1, ..., x_n);
    While (S ≠ ∅) {
        Output min(S);
        del_min(S);
    }
}
```

מימוש sift_down לערימת מינימום

```
sift_down(first, last) {
for (int r = first; r <= last/2; ){
        if (2r = = last){ /* r has one child at 2*r  */
                if (a[r] > a[2*r])
                        swap(r,2*r);
        }
        else { /* r has two children at 2*r and 2*r+1  */
                if (a[r]  >  a[2*r] && a[2*r] <=  a[2*r+1]){
                        swap(a+r, a+2*r);
                        r *= 2;
                }
                else if (a[r] > a[2*r + 1] && a[2*r + 1] <= a[2*r]){
                        swap(a+r, a+2*r + 1);
                        r *= 2*r + 1;
                }
                else break;
}    }    }
```

מיון QuickSort

<u>המיון משתמש בשיטת הפרד ומשול:</u>

- בכל שלב ממיינים את המערך $A[l, \dots, r]$.
- נבחר איבר כלשהו מתוך $A[l, \dots, r]$ שיקרא איבר הציר (pivot) ויסומן ב-$A[p]$.
  למשל נבחר $p = l$ או נבחר את p באקראי.
- <u>הפרד</u>: סדר את $A[l, \dots, r]$ כך שכל האיברים הקטנים מאיבר הציר ימצאו בתאים $A[l, \dots, i-1]$ וכל האיברים הגדולים או שווים לו ימצאו בתאים $A[i, \dots, r]$, כאשר איבר הציר נמצא במקום $A[i]$.
- <u>משול</u>: מיין רקורסיבית את המערכים $A[l, \dots, i-1]$ ואת $A[i+1, \dots, r]$.

```
QuickSort(Key *A, int l, int r){
    if (l >= r) return ;
    int p = choose_pivot(A,l,r);
    int i = partition(A,l,r,p);
    QuickSort(A, l, i-1);
    QuickSort(A,i+1,r);
}
```

<u>הפרד</u>:

<u>משול</u>:

השגרה partition מחזירה את האינדקס בו שמור איבר הציר אחרי הפעלתה.

QuickSort (המשך)

```
QuickSort(Key *A, int l, int r){
    if (l >= r) return ;
    int p = choose_pivot(A,l,r);
    int i = partition(A,l,r,p);
    QuickSort(A, l, i-1);
    QuickSort(A,i+1,r);
}
```

<u>נכונות – בקצרה:</u>
איבר הציר נמצא במקום המתאים לו במערך אחרי ביצוע Partition. הקריאות הרקורסיביות ממיינות את האיברים שאחרי איבר הציר ולפני איבר הציר.

$<p$	$<p$	$<p$	p	$\geq p$	$\geq p$	$\geq p$	$\geq p$	$\geq p$	$\geq p$

<u>זמן המיון - תלוי באיבר הציר. נראה בהמשך:</u>
1. במקרה הגרוע ביותר נקבל זמן $\Theta(n^2)$.
2. אם בכל שלב המערך נחצה בצורה קרובה לשווה, אז נקבל זמן $O(n \log n)$.
3. אם איבר הציר נבחר באקראי, נקבל זמן ריצה של $O(n \log n)$ בממוצע.

כלומר האטרקטיביות של אלגוריתם זה אינה נובעת מזמן הריצה המקסימלי.

Partition - פרוצדורת ההפרדה לפי איבר ציר

```
int partition( KEY *A, int left, int right, int pivot){
	swap (A+pivot, A+right);
	int i = left;
	for (int j=left; j < right; j++) {
		if (A[j] < A[right]) {
			swap(A+i, A+j);
			i++;
		}
	swap(A+right, A+i);
	return i;
}
```

דוגמא: pivot=left.
איבר הציר הוא המספר 7.

7	11	12	6	9	1	3	2	8	10
10	11	12	6	9	1	3	2	8	7
6	11	12	10	9	1	3	2	8	7
6	1	12	10	9	11	3	2	8	7
6	1	3	10	9	11	12	2	8	7
6	1	3	2	9	11	12	10	8	7
6	1	3	2	7	11	12	10	8	9

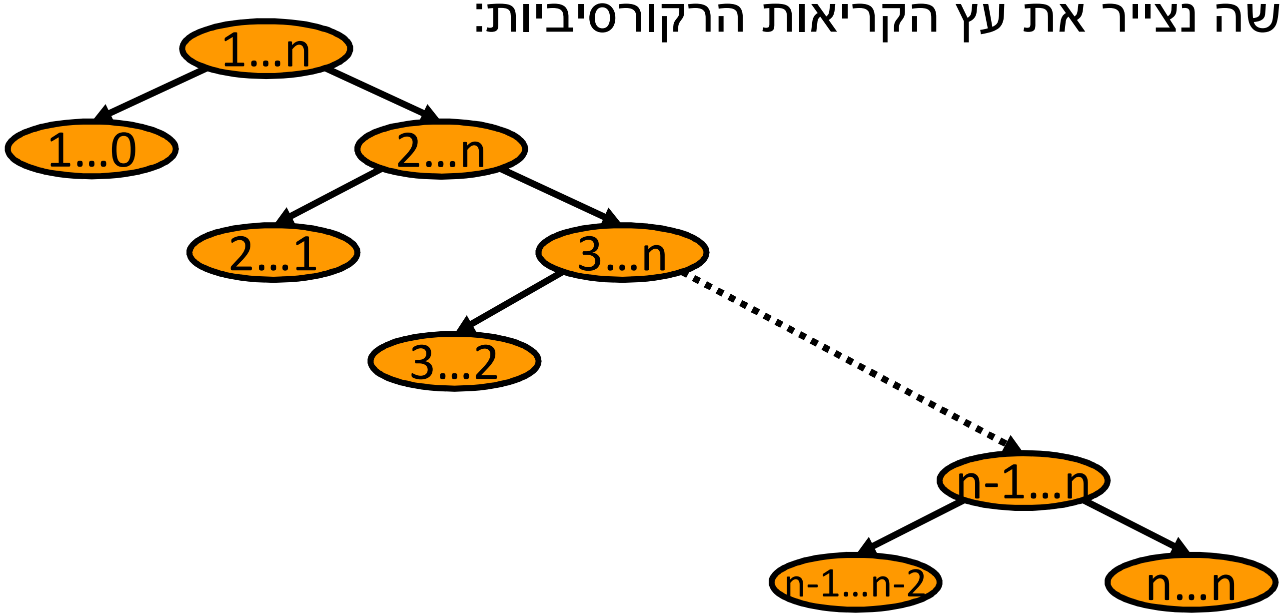
העברת איבר הציר.

זמן ריצה: $\Theta(n)$.

i

ניתוח זמנים במקרה הגרוע

נתון מערך ממוין מהקטן לגדול. נניח שאיבר הציר נבחר להיות האיבר הקטן ביותר.
בכל קריאה רקורסיבית המערך $A[l,n]$ מתבצעת קריאה רקורסיבית למיון המערך $A[l,l-1]$ והמערך $A[l+1,n]$.
לשם המחשה נצייר את עץ הקריאות הרקורסיביות:



<u>זמן הריצה:</u>

$$T(n) = cn + T(n-1) = cn + c(n-1) + T(n-2) =$$
$$= c\sum_{i=1}^{n} i = \Theta(n^2)$$

ניתוח זמנים

זמן הריצה תלוי באיבר הציר.

- מקרה אופטימלי. איבר הציר הוא חציון.

$$T(n) = 2 \cdot T(n/2) + \Theta(n)$$
$$T(n) = \Theta(n \log n)$$

- חלוקה מאוזנת. איבר הציר מחלק את המערך כך שכל חלק יכיל אחוז מסוים מגודל המערך (למשל 10%).

$$T(n) = T(n/10) + T(9n/10) + \Theta(n)$$
$$T(n) = \Theta(n \log n)$$

- מקרה ממוצע. איבר הציר הוא אקראי.

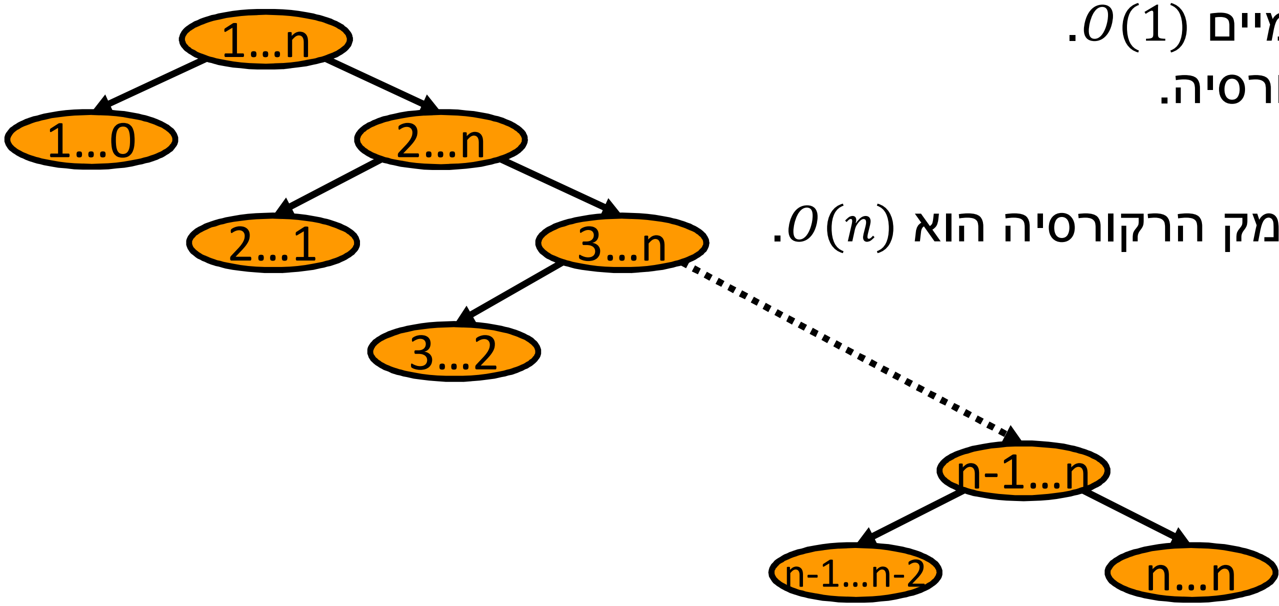
$$T(n) = \Theta(n) + \frac{1}{n}\left(\sum_{i=1}^{n} T(i-1) + T(n-i)\right) \text{ on average}$$
$$T(n) = \Theta(n \log n) \text{ on average}$$

עץ הקריאות הרקורסיביות הוא עץ בינרי ממוצע. נוסחת הנסיגה זהה לנוסחת הנסיגה לזמן בניית עץ חיפוש בינרי אקראי ומכאן זמן הריצה של Quicksort הוא $O(n \log n)$ בממוצע.

ניתוח מקום

<u>המקום שדרוש ל QuickSort כולל:</u>

- מערך הקלט.
- משתנים מקומיים $O(1)$.
- מחסנית הרקורסיה.

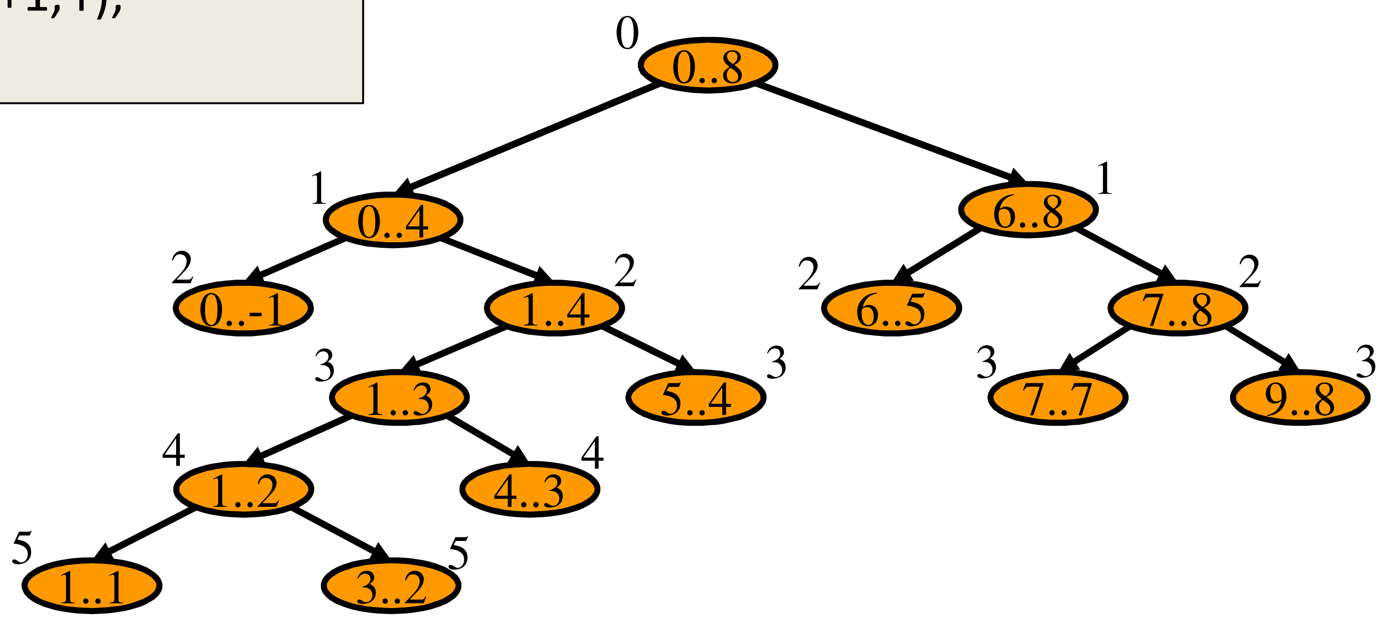


במקרה הגרוע עומק הרקורסיה הוא $O(n)$.

נראה כעת שיטה להגבלת עומק הרקורסיה ל- $O(\log n)$ במקרה הגרוע.

דוגמא להתנהגות מחסנית הרקורסיה

```
QuickSort(Key *a, int l, int r) {
    if (l >= r) return ;
    int p = choose_pivot(a,l,r);
    int i = partition(a,l,r,p);
    QuickSort(a, l, i-1);
    QuickSort(a, i+1, r);
}
```

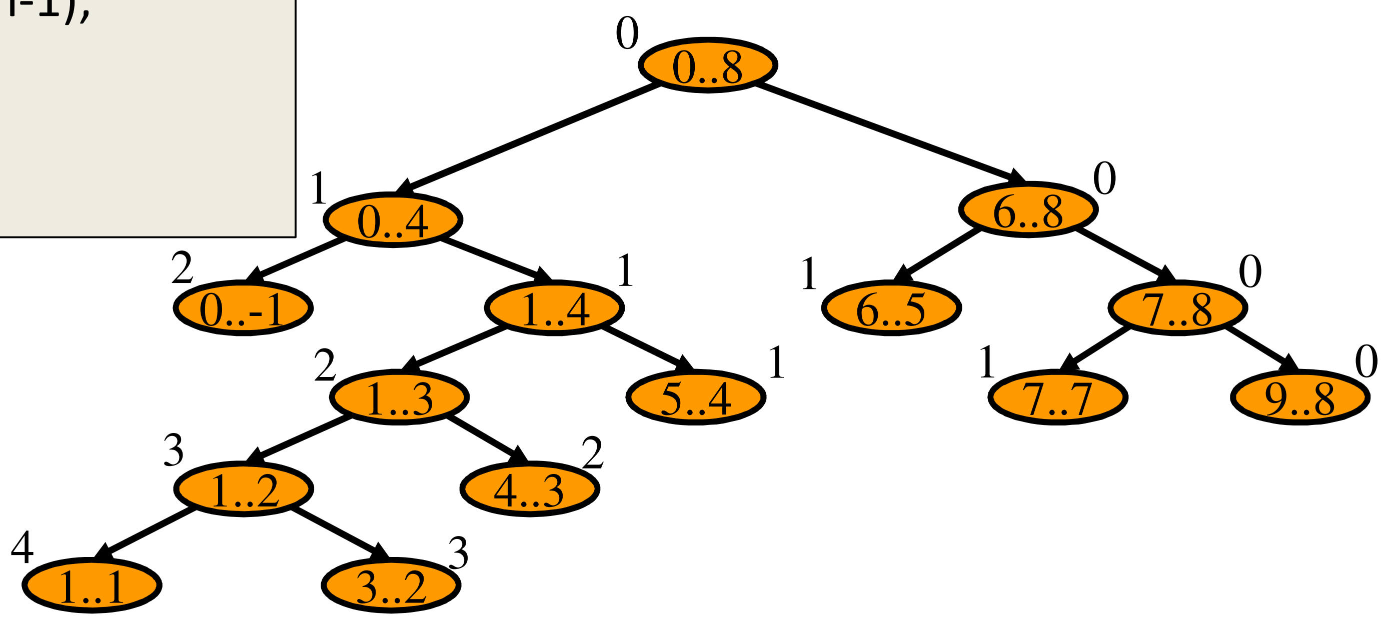


בכל צומת מצוין טווח האינדקסים ולצד הצומת מצוין עומק המחסנית.

העלמת רקורסיית זנב
Tail recursion elimination

```
QuickSort(Key *a, int l, int r) {
    while (l < r){
        int p = choose_pivot(a,l,r);
        int i = partition(a,l,r,p);
        QuickSort(a, l, i-1);
        l = i + 1;
    }
}
```

החלפת הקריאה הרקורסיבית השניה באיטרציה:



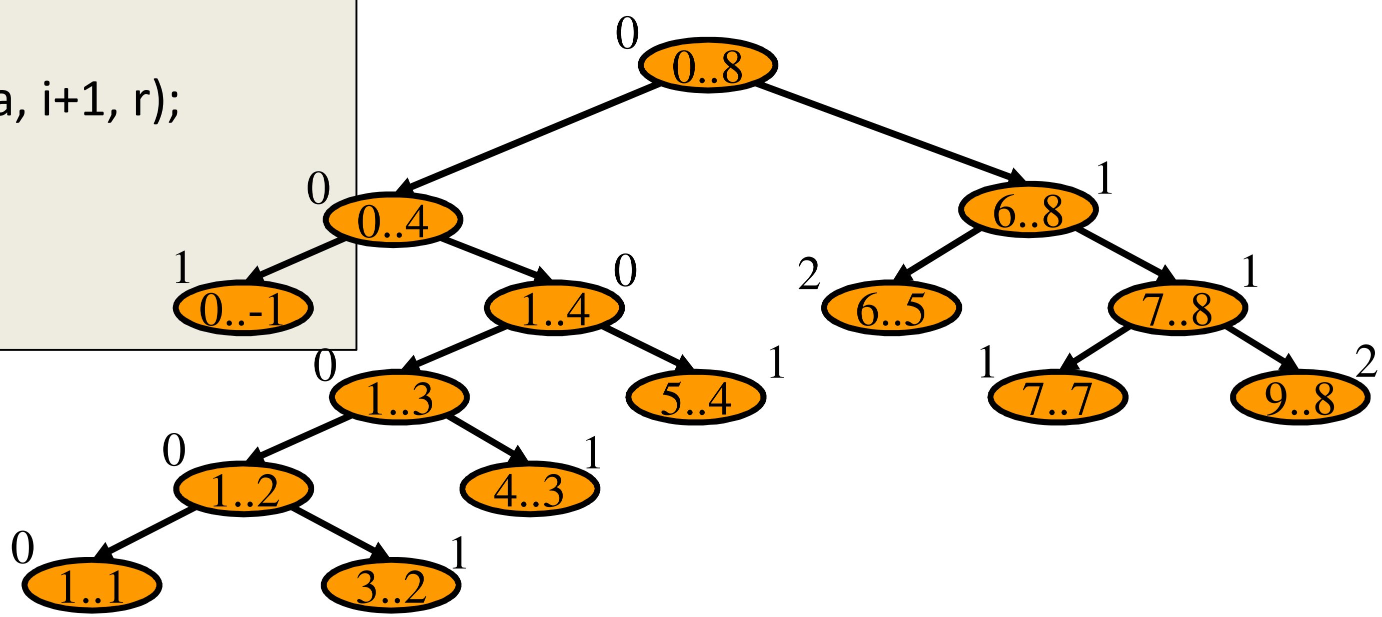
בקומפיילרים מודרניים מימוש של העלמת רקורסיית זנב נעשה אוטומטית.

העלמת רקורסיה לפי גודלה

```
QuickSort(Key *a, int l, int r) {
    while (l < r){
        int p = choose_pivot(a,l,r);
        int i = partition(a,l,r,p);
        if (i - l < r – i) {
            QuickSort(a, l, i-1);
            l = i + 1;
        }
        else {
            QuickSort(a, i+1, r);
            r = i - 1 ;
        }
    }
}
```

החלפת הקריאה הרקורסיבית <u>הגדולה</u> באיטרציה. כלומר, קריאה רקורסיבית רק על חלק המערך הקטן.

<u>התנהגות המחסנית על הדוגמא הקודמת (בבחירה זהה של איבר הציר):</u>



במימוש זה עומק הרקורסיה חסום ע"י $\log_2(n)$ כיוון שבכל קריאה רקורסיבית המערך קטן בלפחות חצי.

סיכום השיעור

<u>בשיעור זה ראינו:</u>

1. <u>ערימה(תור עדיפויות)</u> – מבנה נתונים חדש ומימוש של מבנה זה.
2. <u>Heapsort</u> - שימוש בערימה למטרת מיון ללא זיכרון נוסף בזמן $O(n\log n)$.
3. <u>Quicksort</u> והוכחת זמן ריצה $O(n\log n)$ בממוצע.
4. <u>העלמת רקורסיה</u> – שיטה להקטנת עומק מחסנית הרקורסיה עבור תוכניות רקורסיביות.